2014 47th Annual IEEE/ACM International Symposium on Microarchitecture

B-Fetch: Branch Prediction Directed Prefetching for Chip-Multiprocessors

David Kadjo*, Jinchun Kim*, Prabal Sharma†, Reena Panda‡, Paul Gratz* and Daniel Jimenez§
*Electrical and Computer Engineering, §Computer Science and Engineering, Texas A& M University
†Samsung Austin R&D; ‡ Electrical and Computer Engineering, University of Texas at Austin
{dkadjo, cienlux}@tamu.edu, prabalsharma@gmail.com, reena.panda@utexas.edu
pgratz@gratz1.com, djimenez@cse.tamu.edu

***Abstract*—For decades, the primary tools in alleviating the "Memory Wall" have been large cache hierarchies and data prefetchers. Both approaches, become more challenging in modern, Chip-multiprocessor (CMP) design. Increasing the last-level cache (LLC) size yields diminishing returns in terms of performance per Watt; given VLSI power scaling trends, this approach becomes hard to justify. These trends also impact hardware budgets for prefetchers. Moreover, in the context of CMPs running multiple concurrent processes, prefetching accuracy is critical to prevent cache pollution effects. These concerns point to the need for a light-weight prefetcher with high accuracy. Existing data prefetchers may generally be classified as low-overhead and low accuracy (Next-n, Stride, etc.) or high-overhead and high accuracy (STeMS, ISB). We propose *B-Fetch*: a data prefetcher driven by branch prediction and effective address value speculation. *B-Fetch* leverages control flow prediction to generate an expected future path of the executing application. It then speculatively computes the effective address of the load instructions along that path based upon a history of past register transformations. Detailed simulation using a cycle accurate simulator shows a geometric mean speedup of 23.4% for single-threaded workloads, improving to 28.6% for multi-application workloads over a baseline system without prefetching. We find that *B-Fetch* outperforms an existing "best-of-class" light-weight prefetcher under single-threaded and multiprogrammed workloads by 9% on average, with 65% less storage overhead.**

***Keywords*-Chip-Multiprocessors; Prefetching; Bfetch; Data Cache; Branch Prediction**

I. Introduction

The large gap between processor and memory speed, known as the "Memory Wall" [29], has been studied extensively over the last two decades. Although processor frequency is no-longer on the exponential growth curve of the late 1990's and early 2000's, the drive towards ever greater DRAM capacities and off-chip bandwidth constraints have kept this gap from closing significantly. In addressing the Memory Wall, current architectures mostly focus on two techniques: increasingly large, multi-level cache hierarchies and data prefetching; both approaches, however, become more challenging in modern chip-multiprocessor (CMP) design. With the end of Dennard scaling [5], growing cache size comes at an increasingly high cost in terms of power/energy consumption. As argued by Esmaeilzadeh, *et al.* [8], energy consumption must be justified by increased performance to be practical as VLSI scaling continues; under this constraint the diminishing performance gains seen with increasing cache size become hard to justify relative to their energy cost. These trends also impact hardware budgets for prefetchers. Moreover, in the context of CMPs running multiple concurrent processes, prefetching accuracy is critical to prevent cache pollution effects [7], [28]. Thus, there is a clear need for a light-weight prefetcher with very high accuracy. In this work we present a novel high-accuracy, low-overhead prefetcher for use in chip multiprocessor designs with shared, last-level caches (LLCs). This prefetcher leverages control-flow speculation, together with effective address value speculation to efficiently provide an accurately predicted stream of future memory references.

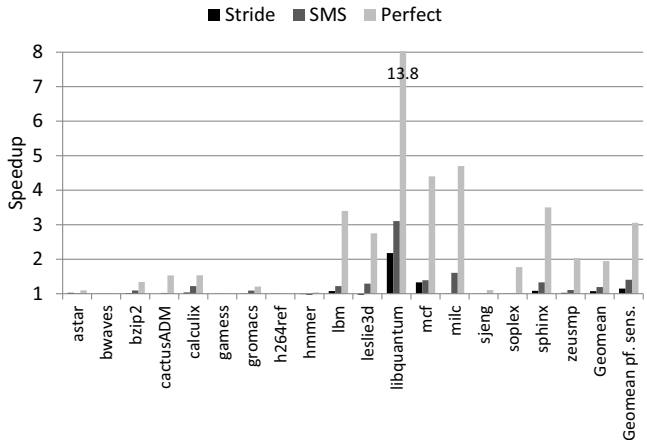


Figure 1: Speedup comparison between the *Stride*, *SMS*, and *Perfect* Prefetchers.

Prefetching is a well known and deeply studied technique in which a hardware mechanism attempts to fill the cache with useful data ahead of the actual demand load request stream coming from the processor. In effect, a perfect prefetcher could make all memory accesses complete as if they were first level cache hits. Figure 1 shows the speedup that might be achieved under such a *Perfect* L1 D-cache prefetcher normalized against the same system without prefetching for a set of SPEC CPU2006 benchmarks. The *Perfect* prefetcher achieves a geometric mean speedup of ∼2X versus no prefetching. For comparison, we also show two other current, light-weight prefetchers, *Stride* [4] and *SMS* [23]. We see that while these prefetchers can provide a significant performance gain, there is room for improvement.

1072-4451/14 $31.00 © 2014 IEEE
DOI 10.1109/MICRO.2014.29

We note that in the figure, several benchmarks see no gain from the perfect prefetcher, these benchmarks are largely L1 cache resident. To focus on the benefit that can be provided with prefetching, we also show the mean across these *prefetch sensitive* benchmarks, denoted as *geomean pf. sens.*.

Many data prefetching techniques have been proposed over the years, however, most existing prefetchers predict future accesses based on current cache misses. For example, sequential prefetchers prefetch the lines sequentially following the current miss [21], stride prefetchers prefetch lines that exhibit a strided pattern with respect to the current miss [4], and region-based prefetchers prefetch a set of blocks around the miss [23]. These techniques are light-weight, energy-efficient and work well for regular memory accesses, however, they tend to be inaccurate for applications with irregular access patterns. More recent prefetchers attempt to address prefetch accuracy for irregular access patterns [22], [11]. While these methods show significantly improved accuracy, they come at a very high cost in storage overhead, either requiring huge structures to record the memory access patterns or the reservation of large amount of off-chip memory for meta-data storage (and the associated, energy consuming shuttling of large meta-data information on and off chip). Under modern constraints on energy and power consumption, it is critical to design efficient, low-overhead prefetching techniques which can address irregular access patterns.

To improve both efficiency and accuracy, we propose to use control flow speculation to feed a prefetch engine. Control flow, *i.e.*, which basic block of instructions is executed and in what sequence, is determined by the direction of branch instructions contained in the path. Each basic block contains a particular set of loads and stores; so, branch instructions directly determine the access patterns of data and can be used to inform a suitable prefetcher. A challenge with this approach is to determine the effective address of a speculative future load given that the register values it is based on are likely to change before the corresponding memory instruction is executed. Our approach builds on the fact register content varies in a predictable manner across a set of basic blocks, even in the case of irregular control flow. Thus it is possible to stitch together the expected transformations of a register across a sequence of predicted basic blocks leading to a given future load instruction. The effective address of that load can then be predicted accurately based on the current architectural state of the register and those predicted transformations. Unlike prior light-weight prefetchers based on current cache misses, this approach has the added advantage that prefetches can be issued for future loads without waiting for an actual miss to occur. Unlike techniques which speculatively continue execution beyond long latency loads [6], [16], our approach is extremely light-weight, with only a small prefetch engine active during operation, rather than the entire core.

In this paper we propose *B-Fetch*, a combined control-flow and effective address speculating prefetching scheme. *B-Fetch* leverages the high prediction accuracy of current-generation branch predictors, combined with a novel effective address speculation technique to identify prefetch candidates. The contributions of this paper are as follows:

- We demonstrate that future memory instruction effective addresses are predictable based on a speculative control flow path from a simple branch predictor. This speculative control flow path is used to feed our *B-Fetch* prefetch engine.
- We propose an effective address value speculation technique based on the current architectural state with learned, per-basic-block variations, to generate effective addresses for the *B-Fetch* prefetch engine.
- We introduce a per-load filtering mechanism to reduce the cache pollution. This technique builds up a confidence estimation for load instructions to determine whether a prefetch candidate is useful or not.
- We show that *B-Fetch* outperforms the best-in-class light-weight prefetcher, Spatial Memory Streaming (*SMS*) [23] by 3.5% for single-threaded workloads (8.5% among prefetch sensitive), and up to 8.9% for multi-application workloads, with 65% less storage overhead than *SMS*.

We evaluate our technique on a set of SPEC CPU2006 benchmarks for both single threaded and multiprogrammed workloads and show performance improvement of 23.4% - 31.2% on average versus baseline. Comparison to the state of the art, light-weight prefetcher shows that *B-Fetch* outperforms it while maintaining a low storage overhead. This paper is organized as follows. Section II discusses the motivation for our technique. Section III reviews previous work in prefetching and value speculation. The design and implementation details of the *B-Fetch* prefetching scheme are presented in Section IV. In Section V, we discuss our experimentation technique and evaluates *B-Fetch*'s performance versus prior techniques. Finally, Section VI concludes the paper.

II. Motivation

Current memory access latencies are of the order of hundreds of processor cycles. To be effective at masking such high latencies, a prefetcher must anticipate misses and issue prefetches far ahead of actual execution. This requires accurate prediction of a) the likely memory instructions to be executed, and b) the likely effective addresses of these instructions.

The program execution path (*i.e.* which basic blocks are executed and in what sequence) is determined by the direction taken by the relevant control instructions. Memory access behavior can therefore be linked to prior control flow behavior. For example, consider the assembly code in Figure 2, consisting of a set of basic blocks and control flow instructions (branches). The basic block executed

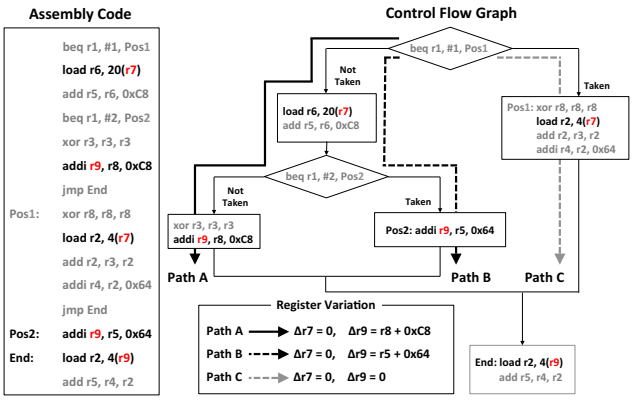
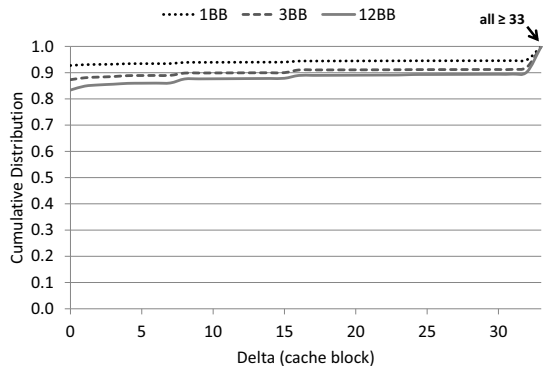


Figure 2: An assembly code fragment and its control flow graph equivalent.

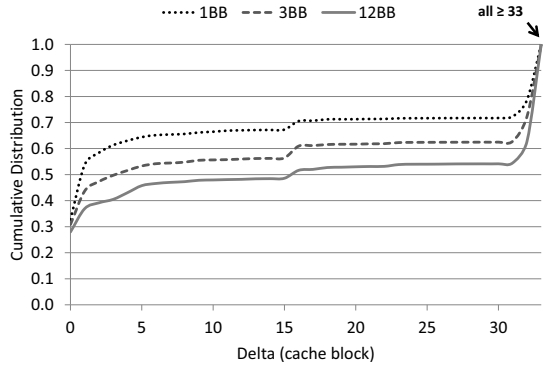
following each control instruction depends on the direction taken by that control instruction. Data requested in future execution phases and its access patterns are dependent on the branch outcomes encountered along the path and the per-block register transformations along that path. We therefore propose a lookahead mechanism that predicts the likely path of execution starting from the current non-speculative branch and issues prefetches for the memory references down that path.

To accurately predict effective addresses down the predicted path, we leverage the observation that each memory reference always uses a particular register for effective address computation. Unlike previously proposed prefetching approaches that use history based effective address computation techniques, we propose to associate register indices being used by the memory instructions with their preceding control instructions (the entry points of the basic block) and use this correlation to identify prefetch candidate addresses. For example, consider the **Path C** illustrated in Figure 2. Relevant memory instructions and their source registers are highlighted in the figure. In **Path C**, both register r7 and r9 do not change their contents. Thus, if we look ahead along the execution path, the effective memory addresses for load instructions can be predicted by adding up static offsets and register values. The lookahead process can be easily performed leveraging support from the branch predictor in the main pipeline. Meanwhile, if the branch is predicted to take **Path A**, the register content of r9 gets changed by the *addi r9, r8, 0xC8* instruction along this path. In this case, if we record the variation of r9 from preceding branch instructions, the effective address can be calculated by adding up current register value, register variation captured in the previous branches, and static offset value. The novelty of *B-Fetch* lies in exploiting branch prediction and predictable register variation to generate the effective memory address.

B-Fetch is based on the premise that register values at the time of effective address generation are correlated in a



(a) Variation of registers content across execution basic block (BB) expressed in granularity of cache block (64B)



(b) Variation of effective addresses across execution basic block (BB) expressed in granularity of cache block (64B)

Figure 3: Cumulative distribution of variation in registers content and effective addresses across execution basic blocks.

predictable way from a) their corresponding values at a time when their preceding branch instructions were executed, and b) the transformations that occur to them over the course of the blocks to that point. Figure 3a shows a cumulative distribution of the register variation (delta) across execution basic blocks (BB), for 1 BB, 3 BB and 12 BB. The variation is expressed at the granularity of a cache block (64B). We observe that for a high percentage (92% in case of 1BB) of registers, the variation falls within 64B. Though that percentage decreases for high number of basic blocks, it remains high (89% for 3BB and 82% for 12BB). By contrast, Figure 3b shows the variation of the effective addresses across 1 to 12 BB. Unlike the register content, the effective address varies considerably, particularly as the depth increases to 12 BB. Hence prefetchers which rely upon stable or predictable changes in effective address are less likely to be accurate than those that can incorporate current register values into the prefetch effective address calculation.

III. Related Work

This section describes the related work in cache prefetching as well as other control-flow speculative techniques. We

classify data prefetchers into two classes, *light-weight* and *heavy-weight*. The light-weight prefetchers have low hardware overhead in terms of state required by the prefetcher. They often suffer from relatively low accuracy and performance. On the other hand, the heavy-weight prefetchers provide high accuracy with substantial performance improvement, at the cost of large amounts of off-chip meta-data memory or additional OS and compiler support.

A. Light Weight Prefetchers

Data Prefetching techniques have been explored extensively as a means to tolerate the growing gap between processor and memory access speeds. Two widely used prefetchers are "Next-n Lines" [21] and *Stride* [4], both of which capture $regular$ memory access patterns with very low hardware overhead. The "Next-n Lines" prefetcher simply queues prefetches for the next n lines after any given miss, under the expectation that the principle of spatial locality will hold and those cache lines after a missed line are likely to be used in the future. The stride prefetcher is slightly more sophisticated; it attempts to identify simple stride reference patterns in programs based upon the past behavior of missing loads. Similar to the "Next-n Lines" technique, when a given load misses, cache lines ahead of that miss are fetched in the pattern following the previous behavior in the hope of avoiding future misses. Both prefetchers have been widely used due to the simple design and low hardware overhead. However, without further knowledge about temporal locality and application characteristics, these prefetchers cannot do more than detecting and prefetching regular memory access patterns with limited spatial locality.

Somogyi *et al.* proposed one of the current top-performing, practical, low-overhead prefetchers, the Spatial Memory Streaming (*SMS*) prefetcher [23]. *SMS* leverages code-based correlation to take advantage of spatial locality in the applications over larger regions of memory (called spatial regions). It predicts the future access pattern within a spatial region around a miss, based on a history of access patterns initiated by that missing instruction in the past. While the *SMS* prefetcher is effective, it is indirectly inferring future program control-flow when it speculates on the misses in a spatial region. As a result, the state overheads of this predictor can be higher than the others in this class.

Generally, while these light-weight prefetching techniques are quite efficient in terms of storage state versus the performance improvement they provide, they have some disadvantages. In all cases they cannot predict the first misses to a region, and further, they achieve relatively low accuracy for irregular accesses. In the context of chip-multiprocessors with shared LLCs, this low accuracy can even cause performance loss, as inaccurate prefetch streams from one application knock out useful data from another, the "friendly fire" scenario outlined by Jerger, *et al.* [7] and Wu, *et al.* [28].

B. Heavy Weight Prefetchers

To overcome the disadvantages of *SMS*, Somogyi *et al.* proposed an extension called Spatio-Temporal Memory Streaming (STeMS) [22]. STeMS exploits temporal access characteristics over larger spatial regions and finer access patterns within each spatial region to re-create a temporally ordered sequence of expected misses to prefetch. Exploiting both temporal and spatial characteristics, it improves the performance by 3% over the *SMS* scheme. This performance benefit is achieved at the expense of a large storage overhead (on the order of several megabytes). To manage this overhead, STeMS keeps most of the meta-data off-chip at any given time, shuttling it on- and off-chip as the program goes through its execution phases [27].

Roth *et al.* proposed a novel prefetching technique for pointer based data structures which extracts a simplified kernel of the data's pointer reference structure and executes it without the intervening instructions [19]. While this is effective for these types of data structures and uneven memory accesses, it provides no benefit for other types of code. The recently proposed Irregular Stream Buffer (ISB) prefetcher introduced by Jain and Lin [11] also attempts to capture irregular memory access patterns. The key idea of the ISB prefetcher is to use an extra level of indirection to create a new structural address space in which correlated physical addresses are assigned consecutive structural addresses. In doing so, streams of correlated memory addresses are both temporally and spatially ordered in this structural addresses space. Thus, the problem of irregular prefetching is converted to sequential prefetching in structural address space. Although the ISB prefetcher shows reasonable performance improvement with less overhead than STeMS, it still requires 8MB of off-chip storage for off-chip meta-data. In addition to that, ISB sees 8.4% memory traffic overhead due to meta-data accesses which does not occur in light-weight prefetchers.

Generally these heavy-weight prefetching techniques show very high accuracy. However, these advantages come at a high cost in terms of meta-data overheads. In energy/power constrained environments it may not be feasible to implement such prefetchers.

C. Branch Directed and Related Techniques

Prior branch-prediction directed prefetching have focused on simple augmentations to the stride based prefetchers [17], [14], with significantly lower performance than current best of class, light-weight prefetchers (*e.i. SMS*). Although these techniques often accurately speculate on which loads will occur, their performance tends to be poor because they do not accurately speculate on the effective address of those loads. In Tango [17], effective addresses from the last execution of a load are combined with offsets to produce the new expected value, using a technique similar to the traditional value speculation techniques. A key insight and novelty of our technique is, for prefetching, effective address

values can be predicted more accurately based upon their variance from the *current architectural state* at an earlier BBs, as opposed to an offset off the *previously generated effective address*, determined by the last execution of that instruction.

Some early work exists in branch-directed, instruction cache prefetching [26], [18], [3], [25]. Recent work by Ferdman, *et al.*, focusing on server and commercial workloads, shows quite significant gains for instruction cache prefetching [10], [9]. We view these approaches as mostly orthogonal, and potentially complimentary to our data cache prefetching design. In our future work we plan to examine how our path confidence estimation scheme might be used to further improve instruction prefetching.

Our technique bears a passing similarity to Runahead execution-based techniques [16], [6]. Runahead execution effectively functions as a prefetcher by speculatively executing past stalls incurred by long-latency memory instructions. While this approach can be effective at producing a prefetch address stream, it incurs a huge cost in terms of energy. In the presence of a long-latency load, a typical core would idle, once the ROB is full, saving energy. In runahead approaches, the processor continues execution a full speed, potentially wasting significant energy. The *B-Fetch* prefetch pipeline is much smaller and lower complexity than the main pipeline, thus it incurs a much lower energy cost in operation. Furthermore, the prefetch pipeline acts independently and simultaneously with the right-path instructions and need not wait for miss-induced stalls to become active. We are aware of no existing prefetcher design which uses a branch predictor to speculate on control-flow, combined with effective address speculation based upon current architectural state in a light-weight prefetcher design.

Among the numerous existing prefetchers, we compare *B-fetch* with *Stride* and *SMS* prefetchers, two other prefetchers in the *light-weight* class. Both *Stride* and *SMS* prefetchers are considered light-weight because they show substantial performance improvement with reasonable hardware complexity. *SMS* has a more sophisticated design than that of the *Stride*, but its additional hardware cost (approximately 37KB) is low enough to be implemented entirely on-chip.

IV. Proposed Design

B-Fetch is a data cache prefetcher that employs two speculative components. It speculates on a) the expected path through at future BBs, and b) the effective addresses of load instructions along that path. The first speculation is directed by a lookahead mechanism that relies on branch prediction to predict the future execution path. For the second, *B-Fetch* records the variation of register contents at earlier branch instructions and exploits this knowledge to predict the effective address. By making use of the *variation of register values* rather than the *effective address history*, *B-Fetch* can issue useful prefetches even for instructions that exhibit irregular control flow and data access patterns.

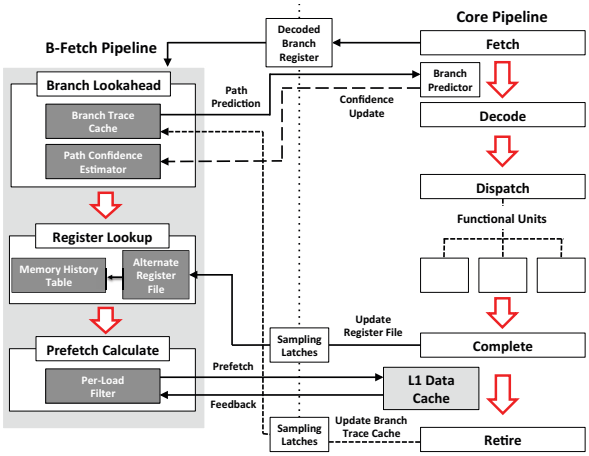


Figure 4: Overall *B-Fetch* microarchitecture.

A. Overview

Figure 4 illustrates the overall system architecture of a *B-Fetch* enabled out-of-order processor. It shows the main CPU execution pipeline and the additional hardware for the *B-Fetch* prefetcher. In our baseline design, the main processor has an out-of-order pipeline with a 4-wide issue width. The *B-Fetch* hardware forms a separate, 3-stage prefetch pipeline parallel to the main pipeline. The *B-Fetch* pipeline is connected to the core's Fetch stage via the Decoded Branch Register (DBR). As branch instructions are decoded in the main execution pipeline, the corresponding PCs are placed into the DBR to initiate the prefetching process. After branch PCs and target addresses are fed into the prefetch pipeline, the *B-Fetch* engine starts to predict future execution path, memory instructions, and their effective addresses. The *B-Fetch* pipeline consists of the following stages:

1) **Branch Lookahead:** This stage is responsible for generating the predicted path of program execution starting from the currently decoded branch. This stage also estimates the confidence along that path, stopping when the confidence falls below a given threshold.
2) **Register Lookup:** This stage is responsible for capturing and providing information about the registers used to generate effective addresses within a given block.
3) **Prefetch Calculate:** This stage is responsible for generating the prefetch addresses that are issued to the prefetch queue, after suitable filtering by a per-load confidence estimator.

B. B-Fetch Microarchitecture

In this section, we outline the detailed description of *B-Fetch* architecture.

1) Branch Lookahead Stage: The first stage of the *B-Fetch* pipeline, the branch lookahead is responsible for accurately predicting the future execution path. Two primary components reside in the Branch Lookahead Stage. First, the predicted future branches and their target addresses

are stored in a small cache called Branch Trace Cache (BrTC). Second, a confidence estimator is used to measure the reliability of the lookahead path and throttle the degree of lookahead when confidence falls below a given threshold.

Starting with a given branch in DBR, each cycle a branch prediction is made and used to look up the PC of the next branch in the BrTC. The confidence of the path to this point is calculated in parallel with the next branch look up. When the confidence falls below a given threshold, the stage will stop predicting further branches.

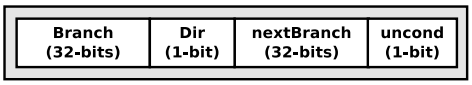


Figure 5: Single Branch Trace Cache (BrTC) entry.

Branch Trace Cache (BrTC): The BrTC captures the dynamic control flow sequence of a program and constructs future lookahead paths across multiple BBs. The intent is to enable jumps from one BB to another, skipping all the non-control-flow changing instructions in between. Figure 5 shows a single BrTC entry, containing the branch address and direction to get to a given BB as well as the branch at the end of that BB. The BrTC is indexed using a hash of the current branch PC, predicted branch direction, and the target address. Thus, a given branch and direction are connected to the next branch in a direct sequential fashion. Each hash used to index the BrTC, containing branch and predicted direction, is also passed on to the Register Lookup stage. To save space, the lower 32 bits of the 43 bit address are used. We found PC branch aliasing in the BrTC to be highly unlikely.

To cover indirect branches, we include a target address to generate a hashed index. In doing so, we allow BrTC to have more flexibility for assigning entries with different targets. The *B-Fetch* pipeline simply borrows the signal from commit stage and uses it to update BrTC. Thus, the BrTC table is dynamically filled in during runtime and only commit-time updates of the entries are allowed.

Path Confidence Estimator: The path confidence estimator controls the depth of lookahead across BBs by keeping track of the cumulative confidence of the predicted paths. Whenever the cumulative path confidence falls below a threshold value, indicating a likelihood of wrong path prediction, the lookahead process is terminated. This technique avoids prefetching useless data by preventing lookahead down a likely mispredicted path. Note that the prefetch control mechanism based on the path confidence has not been explored in any similar prior work. *B-Fetch* calculates the path confidence using a mechanism similar to that proposed by Malik *et al.* [15]. Additionally, we adopt a composite confidence estimator by combining the JRS, up-down and self-counters proposed by Jiménez [12], to estimate individual branch confidence values.

2) Register Lookup Stage: The Register Lookup stage tracks the memory instructions in a given BB and the per-BB transformations to their source register values. A pseudo-architectural state copy of the register file contents is maintained in the Alternate Register File (ARF). Register transformations over BBs are tracked in the Memory History Table (MHT), largest structure in the *B-Fetch* pipeline.

Alternate Register File (ARF): The ARF maintains a copy of the register file contents for use in generating predicted prefetch effective addresses. As illustrated in Figure 4, the ARF is updated with a sampling-latch delayed copy of execution stage generated register values. This approach insures that *B-Fetch* receives timely updates to its ARF, while not being on the critical path of the execution units in main pipeline. Given that the main pipeline is out-of-order, we maintained ARF consistency by only allowing a register to be updated by an instruction younger than the previous instruction that modified it. In the ARF, each register is augmented with an instruction sequence field to keep track of the modifying instruction order. Despite the possibility of speculative, wrong-path updates to the ARF, we found that this approach provided sufficient accuracy for the purpose of generating prefetch effective addresses, with significant improvement in performance versus a retire-stage, purely architectural-state, register file copy.

Memory History Table (MHT): The MHT maintains source register indices, current register values, and offset values to calculate effective addresses for prefetch candidates. Each entry in the table corresponds to a given BB, and is indexed by the hash of the current branch PC, predicted branch direction, and the target address generated in the Branch Lookahead Stage. Figure 6 shows a single MHT entry.

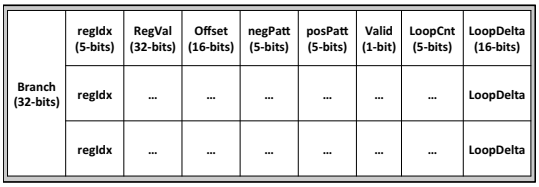


Figure 6: Single Memory History Table (MHT) entry.

Each MHT entry contains a field for the previous *Branch* PC, this is used as a tag to ensure that the hash used to index the entry corresponds to the correct branch. Following, the *Branch* field there are a set of three Register History Entries, each of which containing the following fields: *RegIdx*, *RegVal*, *Offset*, *negPatt*, *posPatt*, *Valid*, *LoopCnt* and *LoopDelta*. A single one of these sets is allocated for each unique register used in generating effective addresses within that basic block. We empirically determined that three Register History Entries was generally sufficient to cover the typical number of registers used in load address computation. More entries consumed more space without

significantly improving performance. Each of these fields within these entries is discussed in the remainder of this section along with their function.

The *RegIdx* holds the source register index used for memory address generation in the corresponding BB, linking source registers to the branch that led to the BB. This linkage is learned as control instructions and memory instructions commit in program order in the main execution pipeline. Figure 3a indicates that the variation of effective addresses across multiple BBs fluctuates more than an offset of the register value. We observe that even if register values and effective addresses do not match exactly, in terms of variation, they still lie within a fixed offset from each other. The *Offset* field retains these fixed-offset relationships between source register values at a given prior branch and the actual effective address generated at the memory instruction. The offset value is learned by monitoring the addresses generated by memory instructions in the main pipeline, compared against the stored *RegVal* field. *B-Fetch* generates the effective prefetching address based on the current value of the linked register (*RegVal*) added with the *Offset* value (see Equation 2). The *RegVal* is read into the MHT from the ARF.

Whenever a memory instruction executes in the main pipeline, the MHT is indexed using the prior branch PC and the *Offset* is updated. It is computed as the difference between the effective address and *RegVal*. Note that the *Offset* value includes not only the static offset from memory instruction but also the variation of register value itself over the course of that BB. The *Valid* bit indicates whether the entry is valid.

$$Offset = [\Delta RegisterValue] + StaticOffset \quad (1)$$

$$PrefetchAddress = [RegisterValue] + Offset \quad (2)$$

Loops: In order to efficiently and accurately prefetch for loops, our prefetching algorithm identifies loops and generates prefetch addresses for their future iterations. We make use of our ability to lookahead across future BBs to allow runtime identification of loops. As an example, consider the code fragment given in Listing 1.

Listing 1: An ALPHA assembly fragment with a loop.

```
Start: load r1, 24(r2)
       lda r2, r2, #128
       cmpeq r2, r3, r1
Br1:   beq r1, Start
```

Assuming that the estimated path confidence is high, the lookahead procedure should yield the following sequence of branch addresses: Br1(Taken) $\longrightarrow$ Br1(Taken) $\longrightarrow$ Br1(Taken), the lookahead depth is determined by the path confidence estimator in the previous stage.

The runtime loop-detection algorithm capitalizes on the idea that if during one complete lookahead process, the same branch is visited more than once, it implies a likely loop in the dynamic instruction stream. The MHT table contains entries to allow the detection and prefetching for loop-based program sequences, allowing dynamic identification of loop based code. The *LoopDelta* field holds the difference between the generated effective addresses over consecutive execution instances of the same instruction. The *LoopCnt* field monitors the iteration count of the loop in the lookahead mode. Equation 3 shows the prefetch effective address generation formula as it is implemented to cover loops.

$$PrefetchAddress = [RegisterValue] + Offset + (LoopCnt \times LoopDelta) \quad (3)$$

Multiple Loads with the same index: The *negpatt* and *pospatt* fields capture the cases when there are, within the same BB, consecutive load instructions off the same source register as a bit vector. Consider a BB with an excerpt snippet of code in Listing 2, both load instructions have the same source register and without any modification. The *pospatt* in this case holds the difference between the static offsets. These fields record, at a granularity of cache block, the difference between the static offset of the load instructions whether negative or positive. This approach avoids duplicating entries in the sets of *RegIdx* for loads off the same source register within the same BB.

Listing 2: Consecutive loads off the same source register.

```
.
load r1, 24(r2)
load r3, 80(r2)
.
```

3) Prefetch Calculate: In this stage, the prefetch effective address is calculated according to Equation 3, using the data pulled out of the MHT and ARF in the previous stage. *B-Fetch* also examines if the associated load address has in the past generated reliable prefetches. If not a filtering mechanism is then used to avoid cache pollution using the Per-load Filter.

Per-load Filter: To avoid cache pollution, wasted bandwidth and energy, it is crucial to reduce the number of useless prefetches. Filtering prefetch requests becomes even more important for systems that prefetch directly into the L1 cache, and those that share an LLC with multiple applications. Conceptually, in *B-Fetch*, the branch confidence mechanism might be thought of as a prefetch filtering mechanism, however in practice we found that even when the branch confidence is high, some loads have effective addresses that are difficult to predict (often within the same BB as others that are predictable). To deal with difficult to predict loads, we implement a per-load PC filtering technique. Our per-load filter measures the confidence of prefetches launched from a given load PC. The filter is inspired by the skewed

Prefetcher	Component	# Entries	Size (KB)
B-Fetch	Branch Trace Cache	256	2.06
	Memory History Table	128	4.5
	Alternate Register File	32	0.156
	Per-Load Prefetch Filter	2048	2.25
	Additional Cache bits	-	1.37
	Prefetch Queue	100	0.51
	Path Confidence Estimator	2048	2
	TOTAL SIZE		**12.84**
SMS	Active Generation Table	64	0.57
	Pattern History Table	16K	36
	TOTAL SIZE		**36.57**

Table I: Hardware storage overhead in KB.

sampling predictor used in prior work to detect cache dead blocks [13].

The per-load filter consists of three different tables which contain 3-bit up-down saturating counters for corresponding prefetch loads. Each table is indexed, using the PC of the load instruction, by different hash function and the counter is incremented when the prefetch address turns out to be accurate. If the prefetch address is inaccurate, the counter is decremented. Each access to the filter yields the sum of three counters and constructs a per-load confidence value. The per-load confidence has precedence over the branch confidence. That is, regardless of current branch confidence, if a per-load confidence falls below to a certain threshold, we stop prefetching for that load PC. In order to implement the per load filtering mechanism, each cache block in the L1D cache is augmented with a 10-bit hash of the load PC for the prefetch address and a 1-bit vector to indicate whether the prefetch is useful. These additional bits are accounted for in our storage overhead.

C. Hardware Cost

The additional hardware storage requirements for *B-Fetch*, as well as *SMS* are summarized in Table I. To optimize *SMS* hardware, we have tested different sizes of spatial regions, accumulation tables, and pattern history tables with the SPEC CPU2006 benchmarks. We observe that the practical *SMS* configuration reported by Somogyi, *et al.* [23] shows the best performance improvement. Unless otherwise specified, we use 2KB spatial regions, a 64-entry accumulation table, and a 16K-entry pattern history table. The filtering table originally proposed by Somogyi, *et al.* is removed because the filtering process can be done by comparing bit vectors in the accumulation table [24]. To optimize storage overhead of *B-Fetch*, we reduce the number of entries in MHT, the largest structure in *B-Fetch*. A sensitivity study for *B-Fetch* with different storage overhead is discussed in the Evaluation section. Because programs typically have more memory instructions than control instructions, a branch-based prefetcher captures the same prefetch candidates at much reduced table sizes.

We assume that the tournament branch predictor in the main pipeline can maintain a limited number of access per cycle equal to the fetch width (4-wide in the baseline design). Seznec *et al.* present a four-way interleaved ALPHA EV8 branch predictor which supports up to 16 branches per cycle, for a branch predictor microarchitecture similar to the one used here [20]. Figure 7 shows the breakdown of the number of branch instructions (both conditional and unconditional) fetched each cycle across 18 SPEC CPU2006 benchmarks. We find that, on average, for more than 99.95% of fetch cycles, a maximum of two branches are fetched per cycle. We observe that fetching four consecutive branch instructions occurs only once per billion instructions. Thus, Figure 7 proves that, most of the time, the branch predictor is available to provide data to *B-Fetch* engine without additional hardware. Should this be deemed prohibitive, it would be trivial to include a copy of the branch prediction hardware for use in prefetching, at the cost of some additional state. In addition to the state elements listed in Table I, a number of small adders and control logic is required, however, these components together are insignificant in area and power consumption relative to the arrays enumerated in the Table.

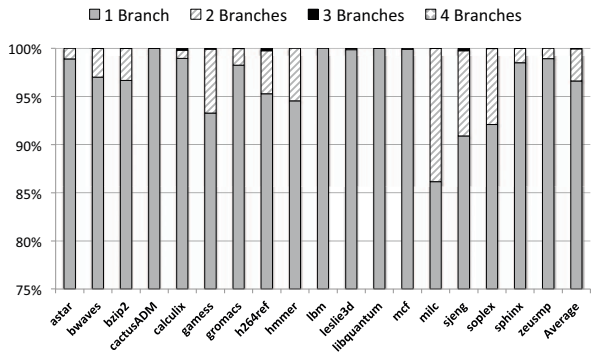


Figure 7: A breakdown of the number of branch instructions fetched per cycle.

CPU	4-wide O3 processor 192-entry ROB
L1I & L1D cache	64KB 8-way 2-cycle latency
L2 cache	Unified 256KB 8-way 10-cycle latency
Shared L3 cache	2MB/Core 16-way 20-cycle latency
Off-chip DRAM	200-cycle latency
Branch predictor	6.55KB Tournament predictor 2.76% branch prediction miss rate
Branch Path Confidence Threshold	0.75
Per-Load Filter Threshold	3

Table II: Baseline configuration.

V. Evaluation

In this section, we discuss the experimental methodology used to evaluate *B-Fetch*. We compare the results obtained versus previous techniques. A sensitivity analysis of our technique is also presented.

A. Methodology

We use gem5 [1], a cycle accurate simulator, to evaluate *B-Fetch*. The baseline configuration is summarized in Table II.

We use the set of 18 SPEC CPU2006 benchmarks our simulation infrastructure currently supports for single-threaded and multiprogrammed workloads simulations. These benchmarks are compiled for the ALPHA ISA. We fast-forward 10 billions instructions, warmup for an additional 1 billion instructions then run in detailed mode for the next 1 billion instructions. For single-threaded workloads, we assume a 2MB LLC. Thus, our result is compared to a baseline with one core and a 2MB LLC. The memory controller bandwidth is limited to 12.8GB/s which is representative of a memory controller of a x64 DDR3. We present the performance as the speedup compared to the baseline configuration ($IPC_{\text{B-Fetch}}/IPC_{\text{baseline}}$). For multiprogrammed workloads simulations, we use the frequency of access (FOA) inter-threads contention model proposed by Chandra *et al* [2] to select 29 mixes of workloads with the highest cache contention. The simulation is stopped when all applications have executed 1 billion instructions. The performance is expressed as the normalized weighted speedup. The weighted speedup is computed as ($\sum(IPC_{\text{multi}}/IPC_{\text{single}})$) , where IPC_{multi} is a workload IPC when executing in the multiprogrammed environment and IPC_{single} the one measured from a single application simulation.

B-Fetch results are compared against two light-weight prefetcher designs, the *Stride* prefetcher and the *SMS* prefetcher, configured as described in Section IV-C. We found that prefetching the next 8 strided addresses to provide the most speedup for a *Stride* prefetcher. We, therefore use such a configuration in our analysis.

B. Results and Analysis

We first present results for *B-Fetch* versus the competing light-weight prefetchers on single-threaded and multiprogrammed workloads. We then present a sensitivity analysis to explore *B-Fetch*'s performance in more detail.

1) Single-threaded Workloads: The speedup for single-threaded workloads is presented in Figure 8. The results are ordered in alphabetical order. We observe that *Stride* performs poorly, compared to other prefetchers, across all the workloads. Therefore, we focus on comparing *B-Fetch* and *SMS*. The *Geomean* column refers to the geometric mean across the entire set of workloads. *B-Fetch* achieves a geometric average speedup of 23.2% compared to 19.7% for *SMS*. *Geomean pf. sens.* refers to the average performance for the prefetch sensitive workloads (*i.e.* those which showed some benefit from the "Perfect Prefetcher" in Figure 1). The results show the *B-Fetch* prefetcher provides a mean speedup of 50.0% across prefetch sensitive benchmarks, compared to 41.5% for *SMS*.

B-Fetch outperforms *SMS* in all but four workloads *cactusADM* , *lbm*, *milc*, and *zeusmp*. Among them, only *milc*

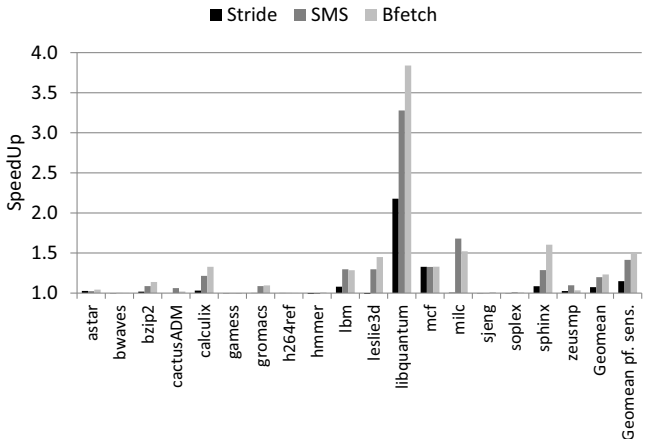


Figure 8: Single-threaded workload speedups.

shows significant performance difference. *milc* is a corner case in that a single miss reference in *milc* tends to be a very strong predictor for a pattern of references within a large spatial region. As a single pattern history table entry in *SMS* covers 2KB spatial region while a comparable entry in *B-Fetch*'s MHT neg/posPatt field covers only 256 bytes. To verify, we evaluated *milc* with smaller spatial regions. With smaller spatial regions, *SMS*'s the performance drops significantly. When the spatial region is set equivalent to the 256 Bytes neg/posPatt size in *B-Fetch*, the performance gain drops to less than 49.23%, less than *B-Fetch*'s 52.15%. For all other memory intensive benchmarks *B-Fetch* shows same or significantly better performance (*lbm*, *leslie3d*, *libquantum*, *mcf*, and *sphinx*). The larger spatial regions are more likely to span unrelated data structures and require more storage overhead to be implemented. Taking the speedup and the cost of storage overhead together, we believe *B-Fetch* presents a better solution for overall data prefetching in single-threaded workloads. We observe that the average lookahead depth is 8 BB with 0.75 branch path confidence.

2) Multiprogrammed Workloads: Figures 9 and 10 show the performance for mixes of 2 workloads (*mix-2*) and mixes of 4 workloads (*mix-4*) respectively. Note that the performance displayed on Figures 9 and 10 is ordered by increasing speedup for *B-Fetch*. In both *mix-2* and *mix-4* configurations, *B-Fetch* achieves a greater speedup as compared to *SMS*. For *mix-2*, *B-Fetch* achieves a speedup of 31.2% compared to 25.5% for *SMS*. Similarly, for *mix-4* *B-Fetch* achieves a performance speedup of 28.5% compared to 19.6% for *SMS*. Preliminary results with mixes of 8 workloads continue this trend.

The improved performance under multiprogrammed workloads, versus single-threaded workloads, is a direct consequence of *B-Fetch*'s improved prefetching accuracy. Jerger, *et. al*, emphasize the harmful effects of prefetching in CMP environments, primarily due to cache and memory bandwidth contention [7]. Since *B-Fetch* provides confidence based control mechanisms (path confidence and per-load

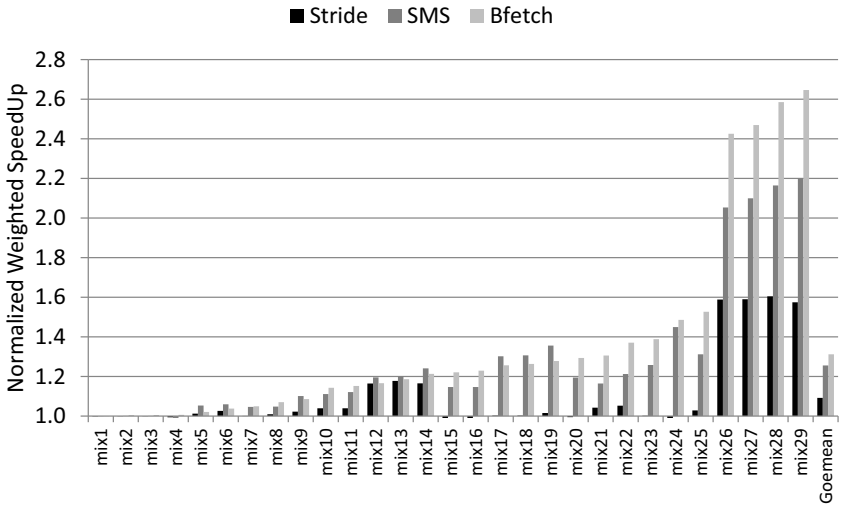


Figure 9: Normalized Weighted Weighted for mixes of 2 workloads.

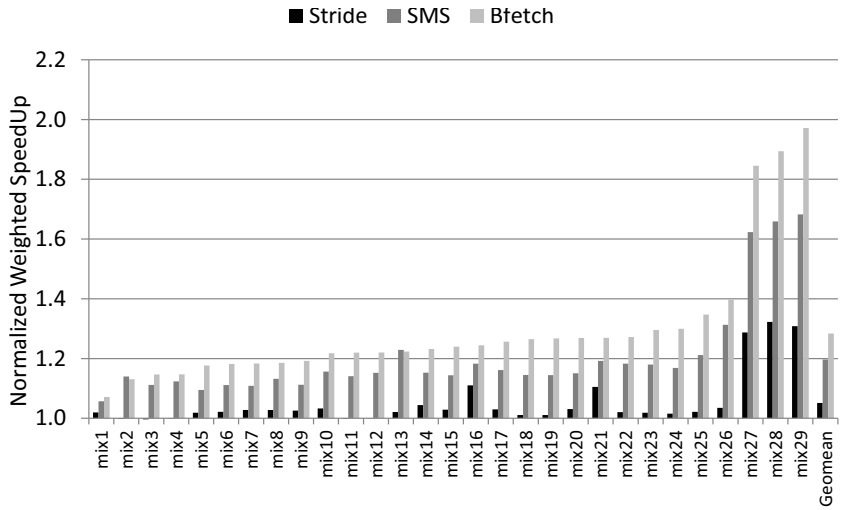


Figure 10: Normalized Weighted Speedup for mixes of 4 workloads.

confidence), it generates less useless prefetch requests. As a result, it causes less pollution in shared LLC than *SMS*. Figure 11 shows the number of useful and useless prefetch issues by *SMS* and *B-Fetch* for all the workloads. We observe that on average *B-Fetch* issues about 4% more useful prefetches while issuing around 50% less useless prefetches compared to *SMS*.

C. Sensitivity Analysis

This section provides a sensitivity analysis of *B-Fetch*. We explore the impact of different parameters and structures on the performance.

1) Branch Confidence: As discussed in Section IV-B, the depth of the lookahead process is controlled by the branch confidence estimation. Figure 12 shows the performance speedup as the branch confidence threshold is varied. The average performance speedup is 20.6%, 23.2% and 23.0% for confidence threshold of 0.45, 0.75 and 0.90 respectively. We observe that the best performance is seen at 0.75. With a lower threshold the increased number of low confidence, potentially wrong-path, prefetches issued, leads to higher cache pollution. We note, however, the speedup difference between confidence 0.45 and 0.75 is not large. Hence, the performance is fairly stable across a range of lookahead confidence depths. This is due in part to the per-load filtering mechanism that is capable of filtering out useless prefetch requests. For the threshold values higher than 0.90, we observe that the lookahead depth decreases. Consequently, the *B-Fetch* engine becomes conservative and prefetches less often.

2) Branch Predictor Size: A branch predictor used in *B-Fetch* is a classic tournament predictor of which final prediction is determined by a simple voting schemes. Thus, we think there might be still room for improvement with

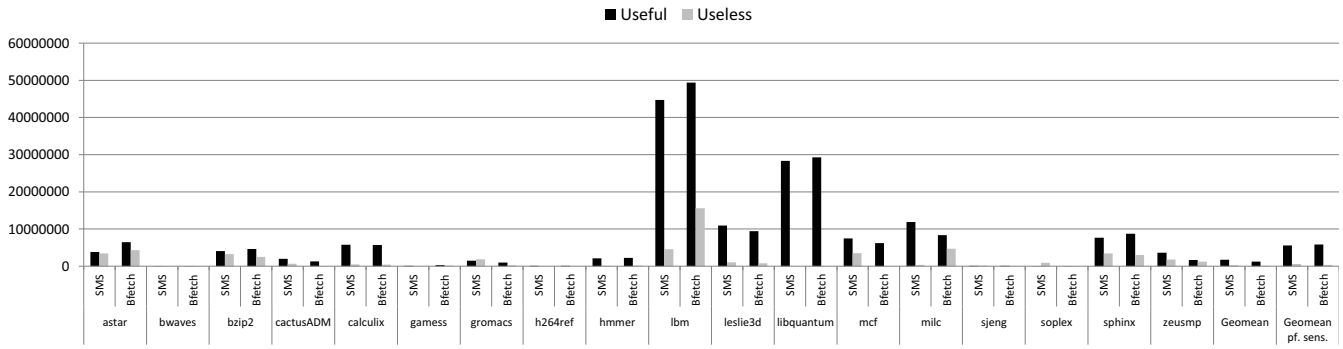


Figure 11: Number of useful and useless prefetches issued.

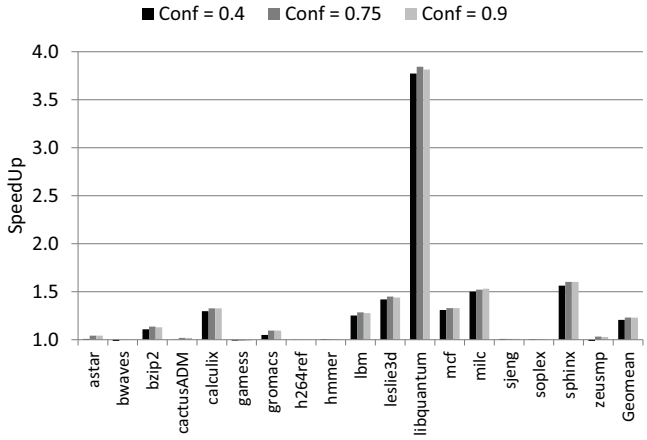


Figure 12: Branch confidence sensitivity.

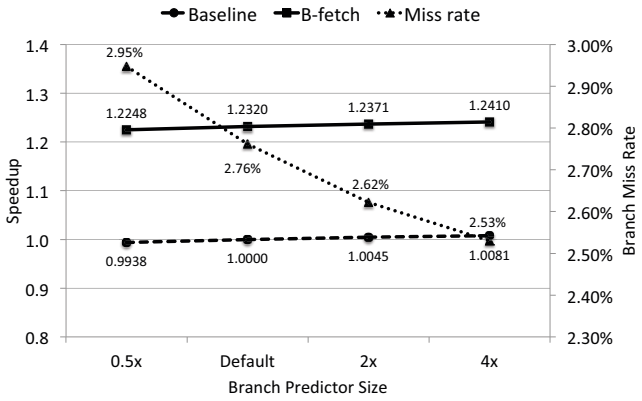


Figure 13: Branch predictor size sensitivity.

more accurate branch predictor. Figure 13 shows the performance variation of baseline and *B-Fetch* enabled processors. To emulate a more accurate branch predictor, we simply increase the size of the tournament branch predictor. The figure also plots the branch miss rate as an arithmetic mean across all benchmark sets. Because the default tournament predictor already provides a very low miss rate for these applications, *B-Fetch* does not show a significant additional performance gain from a larger branch predictor. In the future work, we plan to evaluate *B-Fetch* with the state-of-art branch predictors.

3) CPU Pipeline Width: Figure 14 shows the performance speed up for a 2-wide, 4-wide, and 8-wide out-of-order pipeline. In general, the performance gradually increases as the pipeline width grows. The most considerable performance improvement with wide machine is observed from *leslie3d* and *mcf*. Conversely, the speedup of *libquantum* and *milc* gets saturated at 4-wide machine. The average speedup found was 22.6%, 23.21% and 26.71% for 2-wide, 4-wide, and 8-wide machines respectively, generally indicating that *B-Fetch* provides reasonable speedups across the spectrum from light-weight to heavy-weight cores.

4) B-Fetch *Size:* Since *B-Fetch* lies within the class of *light-weight* prefetchers, it is important to analyze the effects of its storage size on performance. Figure 15 compares the performance improvement for different sizes of *B-Fetch*. The storage overhead is changed to 8.01KB, 9.65KB, 12.94KB and 19.46KB by modifying the number of entries in both the BrTC and the MHT to 64, 128, 256 and 512 respectively. The geometric average speed achieved is 17.0%, 18.9%, 23.21% and 23.1%. We observe the maximum performance speedup is obtained for a size of 12.94KB. Hence the size used for our *B-Fetch* implementation.

VI. Conclusions

This paper proposes *B-Fetch*, a data prefetcher that takes advantage of control flow speculation in the branch predictor to accurately generate data prefetches. *B-Fetch* utilizes the strong correlation between the effective addresses generated by memory instructions and the values of the corresponding source registers at prior branch locations. *B-Fetch* leverages a copy of architectural state of registers at the time of the prefetch together with learned knowledge of the register transformations which occur over BBs to generate accurate and timely prefetches for data exhibiting both regular and irregular access patterns. *B-Fetch* achieves a mean speedup

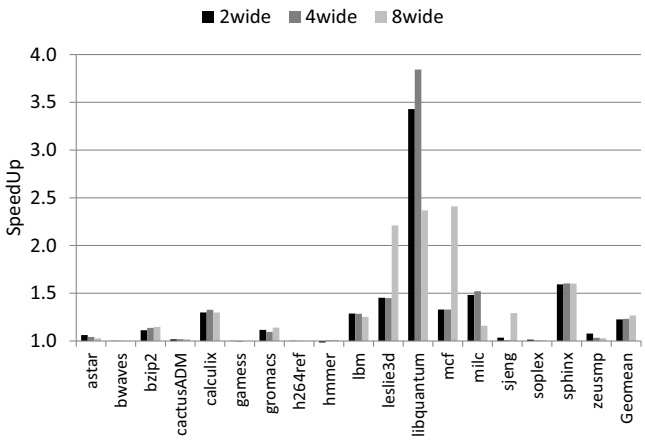


Figure 14: CPU pipeline width sensitivity.

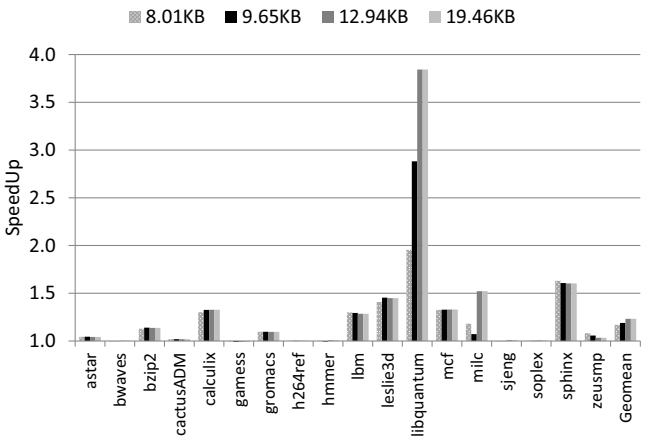


Figure 15: *B-Fetch* storage sensitivity.

of 23.0% over a baseline and outperforms the state-of-the-art prefetcher, while incurring a minimal additional hardware cost.

ACKNOWLEDGMENT

This research is supported in part by the National Science Foundation, under grant CCF-1320074.

REFERENCES

[1] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood, "The gem5 simulator," *SIGARCH Comp. Arch. News*, vol. 39, pp. 1–7, May 2011.

[2] D. Chandra, F. Guo, S. Kim, and Y. Solihin, "Predicting inter-thread cache contention on a chip-multiprocessor architecture," in *HPCA*, 2005, pp. 340–351.

[3] I.-C. Chen, C.-C. Lee, and T. N. Mudge, "Instruction prefetching using branch prediction information," in *IEEE International Conference on Computer Design (ICCD)*, 1997, pp. 593–601.

[4] T. Chen and J. Baer, "Effective hardware-based data prefetching for high-performance processors," *IEEE Transactions on Computers*, vol. 44, pp. 609–623, 1995.

[5] R. H. Dennard, F. H. Gaensslen, V. L. Rideout, E. Bassous, and A. R. Leblanc, "Design of ion-implanted mosfet's with very small physical dimensions," *IEEE Journal of Solid-State Circuits*, vol. 9, pp. 256–268, 1974.

[6] J. Dundas and T. Mudge, "Improving data cache performance by pre-executing instructions under a cache miss," in *The International Conference on Supercomputing (ICS)*, 1997, pp. 68–75.

[7] N. D. Enright Jerger, E. L. Hill, and M. H. Lipasti, "Friendly fire: understanding the effects of multiprocessor prefetches," in *International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2006, pp. 177–188.

[8] H. Esmaeilzadeh, E. Blem, R. S. Amant, K. Sankaralingam, and D. Burger, "Dark silicon and the end of multicore scaling," in *ISCA*, 2011, pp. 365–376.

[9] M. Ferdman, C. Kaynak, and B. Falsafi, in *MICRO*, 2011.

[10] M. Ferdman, T. F. Wenisch, A. Ailamaki, B. Falsafi, and A. Moshovos, "Temporal instruction fetch streaming," in *MICRO*, 2008, pp. 1–10.

[11] A. Jain and C. Lin, "Linearizing irregular memory accesses for improved correlated prefetching." in *MICRO*, 2013, pp. 247–259.

[12] D. A. Jimenez, "Composite confidence estimators for enhanced speculation control," in *The 21st International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*, 2009, pp. 161–168.

[13] S. M. Khan, Y. Tian, and D. A. Jimenez, "Sampling dead block prediction for last-level caches," in *MICRO*, 2010, pp. 175–186.

[14] Y. Liu and D. R. Kaeli, "Branch-directed and stride-based data cache prefetching," in *The International Conference on Computer Design*, ser. ICCD, 1996, pp. 225–230.

[15] K. Malik, M. Agarwal, V. Dhar, and M. Frank, "Paco: Probability-based path confidence prediction," in *HPCA*, 2008, pp. 50–61.

[16] O. Mutlu, J. Stark, C. Wilkerson, and Y. N. Patt, "Runahead Execution: An Alternative to Very Large Instruction Windows for Out-of-Order Processors," in *HPCA*, 2003.

[17] S. Pinter and A. Yoaz, "Tango: a hardware-based data prefetching technique for superscalar processors," in *MICRO*, 1996, pp. 214–225.

[18] G. Reinman, B. Calder, and T. Austin, "Fetch directed instruction prefetching," in *MICRO*, 1999.

[19] A. Roth, A. Moshovos, and G. S. Sohi, "Dependence based prefetching for linked data structures," in *The Eighth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 1998, pp. 115–126.

[20] A. Seznec, S. Felix, V. Krishnan, and Y. Sazeides, "Design tradeoffs for the alpha ev8 conditional branch predictor," in *ISCA*, 2002, pp. 295–306.

[21] A. J. Smith, "Sequential program prefetching in memory hierarchies," *Computer*, vol. 11, pp. 7–21, December 1978.

[22] S. Somogyi, T. F. Wenisch, A. Ailamaki, and B. Falsafi, "Spatio-temporal memory streaming," in *ISCA*, 2009, pp. 69–80.

[23] S. Somogyi, T. F. Wenisch, A. Ailamaki, B. Falsafi, and A. Moshovos, "Spatial memory streaming," in *ISCA*, 2006, pp. 252–263.

[24] S. Somogyi, T. F. Wenisch, M. Ferdman, and B. Falsafi, "Spatial memory streaming," *Journal of Instruction-Level Parallelism (JILP)*, vol. 13, 2011.

[25] L. Spracklen, Y. Chou, and S. G. Abraham, "Effective instruction prefetching in chip multiprocessors for modern commercial applications," in *HPCA*, 2005, pp. 225–236.

[26] V. Srinivasan, E. S. Davidson, G. S. Tyson, M. J. Charney, and T. R. Puzak, "Branch history guided instruction prefetching," in *HPCA*, 2001, pp. 291–300.

[27] T. F. Wenisch, M. Ferdman, A. Ailamaki, B. Falsafi, and A. Moshovos, "Practical off-chip meta-data for temporal memory streaming," in *HPCA*, 2009, pp. 79–90.

[28] C.-J. Wu, A. Jaleel, M. Martonosi, S. C. Steely, Jr., and J. Emer, "Pacman: Prefetch-aware cache management for high performance caching," in *MICRO*, 2011, pp. 442–453.

[29] W. A. Wulf and S. A. McKee, "Hitting the memory wall: implications of the obvious," *SIGARCH Comp. Arch. News*, vol. 23, pp. 20–24, March 1995.